IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Appellant:	Ziegler et al.	Examiner:	Nahar
Serial No.:	09/724,616	Group Art Unit:	2124
Filed:	November 28, 2000	Docket No.:	10001161-1 (HPCO.017PA)
Title:	METHOD AND APPARATUS RESUMING EXECUTION OF A FAILED COMPUTER PROGRAM		

CERTIFICATE UNDER 37 CFR 1.8: The undersigned hereby certifies that this correspondence and the papers, as described hereinabove, are being deposited in the United States Postal Service in triplicate, as first class mail, in an envelope addressed to: Mail Stop Appeal Brief – Patents, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450, on March 15, 2004.

By: Tracey M. Dotter
Tracey M. Dotter

APPEAL BRIEF

Mail Stop Appeal Brief - Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir:

This is an Appeal Brief submitted pursuant to 37 C.F.R. § 1.192 for the above-referenced patent application and is being filed in triplicate.

I. Real Party in Interest

The real party in interest is Hewlett-Packard Company having a place of business at 1501 Page Mill Road, Palo Alto, CA. The above referenced patent application is assigned to Hewlett-Packard Company.

II. Related Appeals and Interferences

Appellant is unaware of any related appeals or interferences.

III. Status of Claims

Claims 1-14 are presented for appeal.

Claims 1 and 9 stand rejected under 35 USC §102(e) as being anticipated by US patent number 6,161,196 to Tsai ("Tsai").

Claims 5, 10, and 11 stand rejected under 35 USC §103(a) over Tsai in view of US patent 5,590,277 to Fuchs et al. ("Fuchs").

Claims 2-4, 6-8, and 12-14 are objected to as being dependent from a rejected base claim but would be allowable if rewritten in independent form including all the limitations of the base claim and any intervening claims.

The claims presented for appeal, as presently amended, may be found in the attached Appendix of Appealed Claims.

IV. Status of Amendments

The application was initially filed on November 28, 2000 and included claims 1-11. In reply to a first Office Action, which was mailed on July 7, 2003, an Amendment and Response was filed on September 2, 2003, *inter alia*, amending claim 10 and adding new claims 12-14 to depend from claim 11. A final Office Action was mailed on November 13, 2003, and in response a Notice of Appeal was filed on January 13, 2004.

V. Summary of Invention

One embodiment of Appellant's invention is directed to a method for software error recovery. Program source code is compiled into a first set of object code with a first compiler (p. 3, ll. 15-25; FIG. 1, 202). The source code is also compiled into a second set of object code with a second compiler (p. 3, ll. 15-25; FIG. 1, 204). Checkpoints are identified in the first and second sets of object code, with each checkpoint in the first set of object code corresponding to a checkpoint in the second set of object code (p. 3, l. 26 – p. 4, l. 9; FIG. 1, 206). Sets of data objects are associated with the checkpoints (p. 3, l. 26 – p. 4, l. 3; p. 5, ll. 10-15; p. 4, ll. 11-23; FIG. 2). Executable checkpoint code is automatically generated for execution at the checkpoints. The checkpoint code is configured to store state information of the associated data objects for recovery if execution of the program is interrupted (p. 3, l. 26

– p. 4, l. 9; FIG. 1, 208). In executing the first set of object code, execution of the checkpoint code stores the state information (p. 3, l. 26 – p. 4, l. 4; p. 5, l. 10-15). Upon detecting an error in execution of the first set of object code, resuming execution of the program using the second set of object code (p. 6, l. 15-30; FIG. 4, 314, 316, 318, 320).

VI. Issues for Review

Issue 1: Is the § 102(e) rejection of claims 1 and 9 proper when the asserted *Tsai* reference fails to teach or suggest every limitation of the claims?

Issue 2: Is the § 103(a) rejection of claims 5, 10, and 11 proper when the asserted *Tsai* and *Fuchs* references fail to teach or suggest all the limitations of the claims, when the rejection fails to cite evidence of motivation, and there is no apparent likelihood of successfully combining the references?

VII. Grouping of Claims

For purposes of this appeal, claims 1 and 9 are in group I, and claims 5, 10, and 11 are in group II. The claims of the different groups do not stand or fall together.

VIII. Argument

Issue 1: The §102(e) rejection of claims 1 and 9 in group I is improper because the asserted *Tsai* reference fails to teach or suggest every limitation of the claims.

The claims of group I include limitations related to compiling program source code into a first set of object code with a first compiler and compiling the program source code into a second set of object code with a second compiler. The first set of object code is executed, and upon detecting an error in execution in the first set of object code, execution is resumed using the second set of object code. Further limitations include generating checkpoint code and storing state information in executing the checkpoint code. The Office Actions fail to show that all these limitations are taught by Tsai.

Generally, Tsai's disclosure describes a system providing fault tolerance (see, e.g., title). A target program is replicated on different machines ("different" as used throughout Tsai is used in the sense of "another" as opposed to "dissimilar"), and each copy of the target program is controlled by a respective backend on a machine. The backends communicate with a frontend, which determines discrepancies between the backends as the copies execute on the machines. The frontend instructs the backends to continue with execution of the copies of the target program if there are no discrepancies (col. 7, ll. 19-48). In the situation where the frontend detects a divergence of results from one of the backends (the "erring" backend) relative to the other backends, the checkpoint data from the non-erring backends is copied to the machine with the erring backend, and the erring backend is reinitiated with the updated checkpoint data (col. 8, ll. 49-61).

The Office Actions are mistaken in the allegation that Tsai's backends inherently contain compilers. As summarized above, in the present invention the program source code is compiled with a first compiler into a first set of object code and compiled with a second compiler into a second set of object code. As explained below, the two compilers are not inherent in Tsai's backends.

MPEP 2112 summarizes the requirements to establish inherency:

> The fact that a certain result or characteristic may occur or be present in the prior art is not sufficient to establish the inherency of that result or characteristic. *In re Rijckaert*, 9 F.3d 1531, 1534, 28 USPQ2d 1955, 1957

> (Fed. Cir. 1993) (reversed rejection because inherency was based on what would result due to optimization of conditions, not what was necessarily present in the prior art); *In re Oelrich*, 666 F.2d 578, 581-82, 212 USPQ 323, 326 (CCPA 1981). "To establish inherency, the extrinsic evidence 'must make clear that the missing descriptive matter is necessarily present in the thing described in the reference, and that it would be so recognized by persons of ordinary skill. Inherency, however, may not be established by probabilities or possibilities. The mere fact that a certain thing may result from a given set of circumstances is not sufficient.' " *In re Robertson*, 169 F.3d 743, 745, 49 USPQ2d 1949, 1950-51 (Fed. Cir. 1999) (citations omitted) (The claims were drawn to a disposable diaper having three fastening elements. The reference disclosed two fastening elements that could perform the same function as the three fastening elements in the claims. The court construed the claims to require three separate elements and held that the reference did not disclose a separate third fastening element, either expressly or inherently.).
>
> "In relying upon the theory of inherency, the examiner must provide a basis in fact and/or technical reasoning to reasonably support the determination that the allegedly inherent characteristic necessarily flows from the teachings of the applied prior art." *Ex parte Levy*, 17 USPQ2d 1461, 1464 (Bd. Pat. App. & Inter. 1990) (emphasis in original).

The Office Actions have not shown that Tsai's backends necessarily include separate compilers that generate separate object code sets.

The most recent Office Action points to the GDB element in backend 28 in Tsai's FIG. 6 in support of the inherency allegation. Tsai explains that the GDB is a debugger process, which controls debugger operations undertaken by the backend. (col. 7, ll. 1-15). A debugger is generally capable of performing many low-level tasks such as managing breakpoints, executing debugger commands when breakpoints are encountered, and printing and modifying the values of variables. (col. 2, l. 66 – col. 5, l. 2). The Office Action is mistaken in reasoning that, each backend must inherently contain "a GCC, a compiler, for the GBD [sic], for the debugger to work."

Even though the Office Actions do not explain and no evidence is provided to describe a GCC, it is assumed that GCC refers to the GNU Compiler Collection (GCC). The website, http://gcc.gnu.org/ explains that the GCC contains front ends for C, C++, Objective-C, Fortran, Java, and Ada, as well as libraries for these languages (emphasis added; see APPENDIX SHOWING GCC WEBSITE in this brief). This clearly indicates that a GCC is a **front** end tool, not a back end component. Furthermore, Tsai clearly shows GDB as a backend tool, and the **APPENDIX SHOWING GDB USAGE** (also in this brief) indicates

that the GDB requires information generated from a compiler. There is no suggestion that the GCC is inherently part of a back end GDB. Indeed, the suggestion is that the front ends and back ends are separate.

As explained in the response to the first Office Action, Tsai may use one compiler to generate object code and then install and run copies of the object code on the various backends. Furthermore, Tsai's description appears to suggest that copies of a program are run (col. 9, l. 64 – col. 10, l. 8), in which case each copy would be from one compiler.

Therefore, the Office Actions fail to show that Tsai's backends inherently include separate compilers that generate separate object code sets.

The claims include further limitations of resuming execution of the program using the second set of object code upon detecting an error in execution of the first set of object code. The Office Actions do not show that these limitations are taught in Tsai's col. 8, ll. 49-61. As this section of Tsai clearly states, the target program on the machine with the erroneous backend is terminated and a new target program is initiated on the machine using checkpoint data. As Tsai later explains, the newly started program is not from a second compiler. Tsai's newly started program is simply a copy of the target program (col. 9, l. 64 – col. 10, l. 8). Thus, Tsai does not resume execution using a second set of object code as claimed.

The Office Actions have not shown that all the limitations of claims 1 and 9 are taught by Tsai. Accordingly, Appellant submits that the §102 rejection is improper and the rejection must be reversed.

Issue 2: The § 103(a) rejection of claims 5, 10, and 11 in group II is improper because the asserted *Tsai* and *Fuchs* references fail to teach or suggest all the limitations of the claims, the rejection fails to cite evidence of motivation, and there is no apparent likelihood of successfully combining the references.

The Office Actions fail to show that the combination teaches or suggests all the claim limitations, fail to provide evidence of motivation for modifying the reference to arrive at the claimed invention, and fail to show a reasonable likelihood that the references could be successfully combined.

The claims of group II include the limitations of claim 1 and further limitations related to selecting between the first and second sets of object code in resuming execution. The Office Actions fail to show a teaching of these limitations by either Tsai or Fuchs. The

explanation provided in the Office Actions is: Fuchs teaches selecting the first set of object code in resuming execution of the program (column 15, lines 48-67 to column 16, lines 1-10), and Tsai teaches selecting the second set of object code (col. 8, ll. 49-61). It is respectfully submitted that the Office Action appears to ignore the claimed aspect of selecting between the first and second sets of object code. No evidence is provided from either reference to suggest that both a first and a second set of object code are considered in making a selection. Furthermore, the Office Action fails to provide evidence that shows either Tsai or Fuchs teaches first and second sets of object code as claimed.

To further show that the Office Action fails to show the limitations of claim 5, the cited sections of Fuchs and Tsai are provided below:

> Fuchs
> The progressive retry recovery algorithm 700 embodying principles of the present invention will be entered at step 701, as shown in FIG. 7. A test is repeatedly performed during step 705 until the error detection monitor 20 of the watchdog 15 detects a fault in a monitored application process. Once a fault is detected during step 705, a test is performed during step 710 to determine if the current detected fault is the same as the previous detected fault for this process. In one embodiment, a fault detected within a predefined time threshold of a previous fault for the same process is assumed to be the same fault. If it is determined during step 710 that the current detected fault is not the same fault as the previous detected fault for this process, if any, then this is the first attempt to remedy this fault, and the progressive retry method should be started at step 1 to attempt to recover the faulty process.
>
> Thus, a counter variable, N, which will control which step of the progressive retry recovery algorithm 700 is currently executing, is set to 1 during step 715, indicating that the first step of the progressive retry recovery algorithm 700 should be attempted. If, however, it is determined during step 710 that the current detected fault is the same fault as the previous detected fault for this process, then previous attempts to remedy this fault have been unsuccessful. Accordingly, the scope of the progressive retry recovery algorithm 700 should be increased in order to attempt to recover the faulty process. Thus, the counter variable, N, is incremented by 1 during step 720, indicating that the next step of the progressive retry recovery algorithm 700 should be attempted. (Fuchs col. 15, l. 48 – col. 16, l. 10).
>
> Tsai
> FIG. 8 illustrates a situation in which the frontend 100 detects a divergence in the reported values from the backends 28-i. The backend with the minority value is identified as the erroneous backend, and execution of the target program is terminated on the corresponding machine. The erroneous backend in this example is backend2 (28-2). A checkpoint is then taken from one of the non-erroneous backends, e.g.,

> backend1 (28-1), and that checkpoint data is copied to the machine with the erroneous backend, i.e., backend2 (28-2), and a new target program is initiated on backend2 using the checkpoint data. In this manner, errors are detected and corrected through a checkpointing recovery mechanism. (Tsai, col. 8, ll. 49-61).

From the cited sections of Fuchs, it appears that Fuchs attempts to recover a faulty process by retrying execution. Tsai appears to restart an erroneous backend with new checkpoint data. Therefore, these cited sections of Fuchs and Tsai make clear that there is no teaching or suggestion of any selecting between first and second object code sets.

The alleged motivation states, "It would have been obvious ... to modify the method disclosed by Tsai to include selecting the first set of object code in resuming execution of the program using the teaching of Fuchs [because] one of ordinary skill in the art would be motivated to debug the faulty program, that is, to recover the faulty program."

No explanation is provided nor is it apparent how debugging a faulty program would be provided by including Fuch's progressive recovery algorithm in Tsai's multi-backend system. Furthermore, this alleged motivation is insufficient because it is merely a broad, conclusory statement of a broadly stated function. The alleged motivation lacks clear and particular reasons that would lead one of ordinary skill in the art to combine specific teachings of Fuchs with Tsai. Addressing the "rigorous ... requirement for a showing of the teaching or motivation to combine prior art references," the Court of Appeals for the Federal Circuit has stated:

> We have noted that evidence of a suggestion, teaching, or motivation to combine may flow from the prior art references themselves, the knowledge of one of ordinary skill in the art, or, in some cases, from the nature of the problem to be solved, (citations omitted), although "the suggestion more often comes from the teachings of the pertinent references," *Rouffet*, 149 F.3d at 1355, 47 USPQ2d at 1456. The range of sources available, however, does not diminish the requirement for actual evidence. That is, the showing must be clear and particular. *See, e.g., C.R. Bard*, 157 F.3d at 1352, 48 USPQ2d at 1232. Broad conclusory statements regarding the teaching of multiple references, standing alone, are not "evidence." (citation omitted) *In re Dembiczak*, 175 F.3d 994, 50 U.S.P.Q.2d 1614 (Fed. Cir. 1999).

The alleged motivation is merely a broad conclusory statement of general applicability, and no evidence is provided to suggest the combination. Therefore, the alleged motivation is insufficient to support *prima facie* obviousness.

The rejection further fails to show that Tsai could be successfully modified with the teachings of Fuchs. For example, Tsai uses modular redundancy to provide fault tolerance (title) apparently because "other conventional schemes use algorithm-based detection methods that are generally not applicable to many types of programs." (col. 1, ll. 61-63). Thus, Tsai appears to teach away from modification using an algorithm such as Fuchs' progressive recovery algorithm. The MPEP states that when a proposed modification would render the teachings being modified unsatisfactory for their intended purpose, there is no suggestion or motivation to make the proposed modification under 35 U.S.C. § 103(a). See MPEP § 2143.01 and In re Gordon, 733 F.2d 900, 221 USPQ 1125 (Fed. Cir. 1984) (A §103 rejection cannot be maintained when the asserted modification undermines the purpose or operation of the main reference.).

For at least the reasons set forth above, *prima facie* obviousness is not established for the claims in group II; the Office Actions fail to show that the combination teaches or suggests all the claim limitations, fail to provide evidence of motivation for modifying the reference to arrive at the claimed invention, and fail to show a reasonable likelihood that the references could be successfully combined. Accordingly, Appellant submits that the §103 rejection is improper and the rejection must be reversed.

The claims in group II are separately patentable over the claims in group I. The claims in group II include limitations that relate to selecting between the first and second object code sets. These limitations are not necessarily present in, nor an obvious extension of the claims of group I. Therefore, the claims in groups I and II are separately patentable.

It is respectfully submitted that in preparing this Appeal Brief, claim 10, being an apparatus claim, was found to have a "selecting" step instead of "means for selecting." For purposes of considering this appeal, the "selecting" should be understood to be "means for selecting". This oversight will be corrected once the application is in an examination stage that permits amendment of the claims.

IX. Conclusion

In view of the above, Appellant believes the claimed invention to be patentable. The Office Action makes incomplete and erroneous findings of fact in the various combinations of references and fails to satisfy the requirements for establishing *prima facie* cases of

anticipation and obviousness. These mistakes are the basis of erroneous conclusions of law pertaining to non-allowability of the claims.

Claims 1-14 remain for consideration. Appellant respectfully requests reversal of the rejections as applied to the appealed claims and allowance of the entire application.

Respectfully submitted,

CRAWFORD MAUNU PLLC
1270 Northland Drive – Suite 390
St. Paul, MN 55120
(651) 686-6633

By: _____
Name: LeRoy D. Maunu
Reg. No.: 35,274

APPENDIX OF APPEALED CLAIMS (09/724,616)

1. A computer-implemented method for software error recovery, comprising:

compiling program source code into a first set of object code with a first compiler;

compiling the program source code into a second set of object code with a second compiler;

identifying checkpoints in the first and second sets of object code, each checkpoint in the first set of object code corresponding to a checkpoint in the second set of object code;

associating sets of data objects with the checkpoints;

automatically generating executable checkpoint code for execution at the checkpoints, the checkpoint code configured to store state information of the associated data objects for recovery if execution of the program is interrupted;

executing the first set of object code;

storing the state information in executing the checkpoint code; and

upon detecting an error in execution of the first set of object code, resuming execution of the program using the second set of object code.

2. The method of claim 1, further comprising:

upon detecting an error in execution of the first set of object code, initially re-executing the first set of object code; and

resuming execution using the second set of object code if the first set of object code fails in re-execution.

3. The method of claim 2, further comprising re-executing the first set of object code a selected number of times before resuming execution using the second set of object code.

4. The method of claim 3, further comprising ceasing resumption of execution of the first and second sets of object code if an error is detected in executing both sets of object code.

5. A computer-implemented method for software error recovery, comprising:

compiling program source code into a first set of object code with a first compiler;

compiling the program source code into a second set of object code with a second compiler;

identifying checkpoints in the first and second sets of object code, each checkpoint in the first set of object code corresponding to a checkpoint in the second set of object code;

associating sets of data objects with the checkpoints;

automatically generating executable checkpoint code for execution at the checkpoints, the checkpoint code configured to store state information of the associated data objects for recovery if execution of the program is interrupted;

executing the first set of object code;

storing the state information in executing the checkpoint code; and

upon detecting an error in execution of the first set of object code, selecting between the first set of object code and the second set of object code in resuming execution of the program.

6. The method of claim 5, further comprising:

upon detecting an error in execution of the first set of object code, initially re-executing the first set of object code; and

resuming execution using the second set of object code if the first set of object code fails in re-execution.

7. The method of claim 6, further comprising re-executing the first set of object code a selected number of times before resuming execution using the second set of object code.

8. The method of claim 7, further comprising ceasing resumption of execution of the first and second sets of object code if an error is detected in executing both sets of object code.

9. An apparatus for software error recovery, comprising:

means for compiling program source code into a first set of object code with a first compiler;

means for compiling the program source code into a second set of object code with a second compiler;

means for identifying checkpoints in the first and second sets of object code, each checkpoint in the first set of object code corresponding to a checkpoint in the second set of object code;

means for associating sets of data objects with the checkpoints; and

means for automatically generating executable checkpoint code for execution at the checkpoints, the checkpoint code configured to store state information of the associated data objects for recovery if execution of the program is interrupted;

means for executing the first set of object code;

means for storing the state information in executing the checkpoint code; and

means for resuming execution of the program using the second set of object code upon detecting an error in execution of the first set of object code.

10. An apparatus for software error recovery, comprising:

means for compiling program source code into a first set of object code with a first compiler;

means for compiling the program source code into a second set of object code with a second compiler;

means for identifying checkpoints in the first and second sets of object code, each checkpoint in the first set of object code corresponding to a checkpoint in the second set of object code;

means for associating sets of data objects with the checkpoints;

means for automatically generating executable checkpoint code for execution at the checkpoints, the checkpoint code configured to store state information of the associated data objects for recovery if execution of the program is interrupted;

means for executing the first set of object code;

means for storing the state information in executing the checkpoint code; and

selecting between the first set of object code and the second set of object code in resuming execution of the program upon detecting an error in execution of the first set of object code.

11. A computer program product configured for causing a computer to perform the steps comprising:

compiling program source code into a first set of object code with a first compiler;

compiling the program source code into a second set of object code with a second compiler;

identifying checkpoints in the first and second sets of object code, each checkpoint in the first set of object code corresponding to a checkpoint in the second set of object code;

associating sets of data objects with the checkpoints;

automatically generating executable checkpoint code for execution at the checkpoints, the checkpoint code configured to store state information of the associated data objects for recovery if execution of the program is interrupted;

executing the first set of object code;

storing the state information in executing the checkpoint code; and

upon detecting an error in execution of the first set of object code, selecting between the first set of object code and the second set of object code in resuming execution of the program.

12. The computer program product of claim 11, further configured for causing a computer to perform the steps comprising:

upon detecting an error in execution of the first set of object code, initially re-executing the first set of object code; and

resuming execution using the second set of object code if the first set of object code fails in re-execution.

13. The computer program product of claim 12, further configured for causing a computer to perform the step comprising re-executing the first set of object code a selected number of times before resuming execution using the second set of object code.

14. The computer program product of claim 13, further configured for causing a computer to perform the step comprising ceasing resumption of execution of the first and second sets of object code if an error is detected in executing both sets of object code.

APPENDIX SHOWING GCC WEBSITE

About GCC

Mission Statement
Mailing lists
Timeline
Contributors
Steering Committee

Documentation

Installation
· Supported Platforms
· Testing
Manual
FAQ
Further Readings

Download

Releases
Snapshots
Mirror sites
Binaries

Development

Development Plan
· (tentative timeline)
Contributing
...Why?
Open projects
Front ends
Back ends
Extensions
CVS read access
Rsync read access

Welcome to the GCC home page!

GCC is the GNU Compiler Collection, which currently contains front ends for C, C++, Objective-C, Fortran, Java, and Ada, as well as libraries for these languages (libstdc++, libgcj,...). Further frontends are available.

Major decisions about GCC are made by the steering committee, guided by the mission statement.

We encourage everyone to contribute changes and help testing GCC, and we provide access to our development sources with anonymous CVS and weekly snapshots.

We strive to provide regular, high quality releases, which we want to work well on a variety of native (including GNU/Linux) and cross targets. To that end, we use an extensive test suite and automated regression testers as well as various benchmark suites and automated testers to maintain and improve quality.

Active release branch: will become GCC 3.4.0
Branch status: 2004-03-09 (open for bug fixes for regressions only).
Current release series: GCC 3.3.3 (released 2004-02-14)
Branch status: 2004-02-21 (open for all bugfixes).
Active development (mainline): will become GCC 3.5.0 (current changes)
Stage 1; open for all maintainers.

News/Announcements

February 25, 2004
The tree-ssa branch has been frozen to be incorporated into GCC 3.5.0. Tree SSA incorporates two new high-level intermediate languages (GENERIC and GIMPLE), an optimization framework for GIMPLE based on the Static Single Assignment (SSA) representation, several SSA-based optimizers and various other improvements to the internal structure of the compiler that allow new optimization opportunities that were difficult to implement before.

February 24, 2004
GCC 3.3.3 has been released.

February 6, 2004
Josef Zlomek of SUSE Labs and Daniel Berlin of IBM Research have contributed Variable Tracking. It generates more accurate debug info about locations of variables and allows debugging code compiled with `-fomit-frame-pointer`.

October 18, 2003

CVSup mirrors
CVS write access

Bugs

Report a bug
Bug database (Bugzilla)
...Management
Known bugs

Get our announcements:
your e-mail Subscribe!

Bernardo Innocenti of Develer S.r.l. has contributed the m68k-uclinux target and improved support for ColdFire cores, based on former work by Paul Dale (SnapGear, Inc.) and Peter Barada (Motorola, Inc.).

October 17, 2003
GCC 3.3.2 has been released.

August 27, 2003
Nicolas Pitre has contributed his hand-coded floating-point support code for ARM. It is both significantly smaller and faster than the existing C-based implementation. The arm-elf configuration uses the new code now, and other ports will follow.

August 8, 2003
GCC 3.3.1 has been released.

June 26, 2003
Ben Elliston of Wasabi Systems, Inc. has converted the existing ARM processor pipeline description to the new DFA pipeline description model. It will be part of the GCC 3.4.0 release.

Older news and announcements...
GCJ news | Fortran 77 news | Fortran 95 status

This search will allow you to search the contents of all the publicly available WWW documents at gcc.gnu.org.
There is also a detailed search form.
Match: All words Type: Exact Limit to:
All pages (including mailing lists and testresults) Sort by: Best Match Search-syntax
Search: Search

Please send FSF & GNU inquiries & questions to gnu@gnu.org. There are also other ways to contact the FSF.

These pages are maintained by the GCC team.

For questions related to the use of GCC, please consult these web pages and the GCC manuals. If that fails, the gcc-help@gcc.gnu.org mailing list might help.
Please send comments on these web pages and the development of GCC to our public developer mailing list at gcc@gnu.org or gcc@gcc.gnu.org.

APPENDIX SHOWING GDB USAGE

4. Running Programs Under GDB

When you run a program under GDB, you must first generate debugging information when you compile it.

You may start GDB with its arguments, if any, in an environment of your choice. If you are doing native debugging, you may redirect your program's input and output, debug an already running process, or kill a child process.

4.1 Compiling for debugging

In order to debug a program effectively, you need to generate debugging information when you compile it. This debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.

To request debugging information, specify the `-g' option when you run the compiler.

Most compilers do not include information about preprocessor macros in the debugging information if you specify the `-g' flag alone, because this information is rather large. Version 3.1 of GCC, the GNU C compiler, provides macro information if you specify the options `-gdwarf-2' and `-g3'; the former option requests debugging information in the Dwarf 2 format, and the latter requests "extra information". In the future, we hope to find more compact ways to represent macro information, so that it can be included with `-g' alone.

Many C compilers are unable to handle the `-g' and `-O' options together. Using those compilers, you cannot generate optimized executables containing debugging information.

GCC, the GNU C compiler, supports `-g' with or without `-O', making it possible to debug optimized code. We recommend that you *always* use `-g' whenever you compile a program. You may think your program is correct, but there is no sense in pushing your luck.

When you debug a program compiled with `-g -O', remember that the optimizer is rearranging your code; the debugger shows you what is really there. Do not be too surprised when the execution path does not exactly match your source file! An extreme example: if you define a variable, but never use it, GDB never sees that variable--because the compiler optimizes it out of existence.

Some things do not work as well with `-g -O' as with just `-g', particularly on machines with instruction scheduling. If in doubt, recompile with `-g' alone, and if this fixes the problem, please report it to us as a bug (including a test case!).

Older versions of the GNU C compiler permitted a variant option `-gg' for debugging information. GDB no longer supports this format; if your GNU C compiler has this option, do not use it.